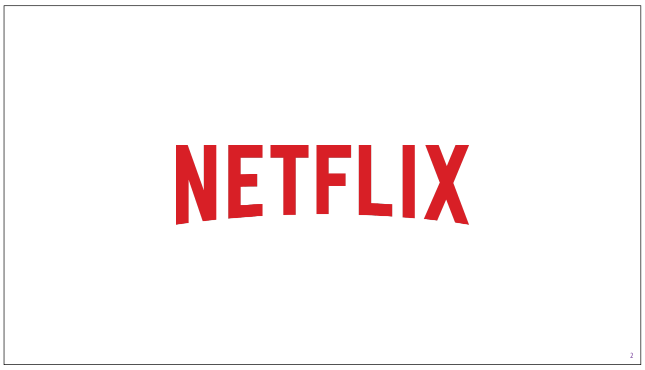
Embedding models
Part 1: Recommenders

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this lecture, we'll take the idea of embedding vectors we first saw in the previous lecture, and we'll look at some other places it can be applied.

Our first topic is a very common task for machine learning: recommender systems. This is something that isn't quite classification or regression and is best modeled as an abstract task in its own right.



NETFLIX

2

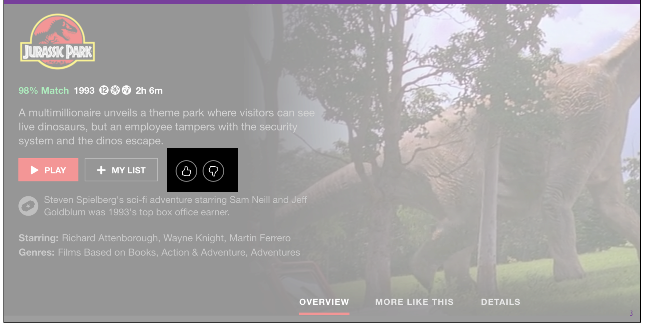
The basic, standard example of a recommender system is recommending **movies** to **users**.

That's no accident. The modern concept of a recommender system was probably born in 2006 when Netflix, then mainly a DVD rental service, released a dataset of user/movie ratings, and offered a 1M$ prize for anybody who could improve the RMSE of their current predicted ratings by 10% from.

This not only sparked an interest in **recommendation** as a task, but also probably started the craze for machine learning competitions that later led to websites like Kaggle.

We'll use the movie task as a running example, but we'll also look at some other settings that translate to the same abstract task.

explicit feedback: users give direct ratings



98% Match 1993 2h 6m

A multimillionaire unveils a theme park where visitors can see live dinosaurs, but an employee tampers with the security system and the dinos escape.

Steven Spielberg's sci-fi adventure starring Sam Neill and Jeff Goldblum was 1993's top box office earner.

Starring: Richard Attenborough, Wayne Knight, Martin Ferrero
Genres: Films Based on Books, Action & Adventure, Adventures

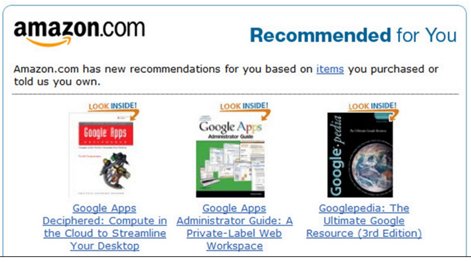
3

Let's start by looking at the Netflix task, with what types of data we have available. The defining property of the abstract task of recommendation is that the primary source of data is **explicit user ratings**: we *ask* users to tell us which movies they like, and hopefully, they'll oblige. They do this only for a few movies, and from the small set of user/movie pairs that we know the rating for, we must predict the rest.

Predicting ratings based on explicit feedback is sometimes known as **collaborative filtering**. The users collaborate by providing ratings, to help filter the movies they'll like out of the large amount of available movies.

The main drawback here is that the information can be very sparse: we'll only get a few ratings per user, and some users won't give any ratings at all. We'll look at some ways to deal with that in the next video. For now, we'll see what we can do with just explicit feedback.

Amazon



amazon.com **Recommended** for You

Amazon.com has new recommendations for you based on items you purchased or told us you own.

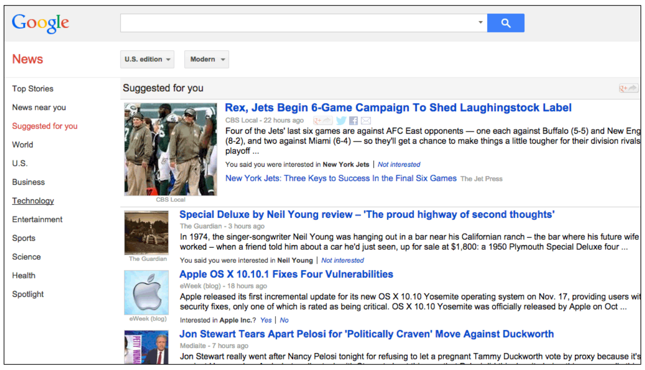
Google Apps Deciphered: Compute in the Cloud to Streamline Your Desktop

Google Apps Administrator Guide: A Private-Label Web Workspace

Googlepedia: The Ultimate Google Resource (3rd Edition)

Movie recommendation is the canonical use case for recommender systems, but the system applies to many other systems.

Amazon was probably the first to use personalised recommendation to help users navigate their website. The principle is similar to Netflix. These are many **users**, a large database of **items**. We have some information about which users liked which items in the past, so we can predict which ones they'll like in the future.



Google News

U.S. edition · Modern

Top Stories
News near you
Suggested for you
World
U.S.
Business
Technology
Entertainment
Sports
Science
Health
Spotlight

Suggested for you

Rex, Jets Begin 6-Game Campaign To Shed Laughingstock Label
CBS Local - 22 hours ago
Four of the Jets' last six games are against AFC East opponents — one each against Buffalo (5-5) and New Eng (8-2), and two against Miami (6-4) — so they'll get a chance to make things a little tougher for their division rivals playoff ...
You said you were interested in **New York Jets** | *Not interested*
New York Jets: Three Keys to Success In the Final Six Games The Jet Press
CBS Local

Special Deluxe by Neil Young review – 'The proud highway of second thoughts'
The Guardian - 3 hours ago
In 1974, the singer-songwriter Neil Young was hanging out in a bar near his Californian ranch – the bar where his future wife worked – when a friend told him about a car he'd just seen, up for sale at $1,800: a 1950 Plymouth Special Deluxe four ...
You said you were interested in **Neil Young** | *Not interested*
The Guardian

Apple OS X 10.10.1 Fixes Four Vulnerabilities
eWeek (blog) - 18 hours ago
Apple released its first incremental update for its new OS X 10.10 Yosemite operating system on Nov. 17, providing users wi security fixes, only one of which is rated as being critical. OS X 10.10 Yosemite was officially released by Apple on Oct ...
Interested in **Apple Inc.**? *Yes* | *No*
eWeek (blog)

Jon Stewart Tears Apart Pelosi for 'Politically Craven' Move Against Duckworth
Mediaite - 7 hours ago
Jon Stewart really went after Nancy Pelosi tonight for refusing to let a pregnant Tammy Duckworth vote by proxy because it's

Another use case is news stories, helping people find the articles they're interested in.

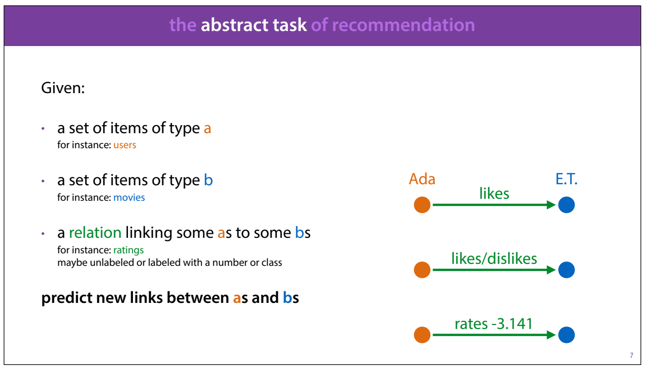
subject	has_property	object
user	liked	movie
user	bought	book
user	rating = -1	movie
recipe	has_ingredient	ingredient
politician	voted_for	law
person	friend_of follows swiped_right	person

In the most general sense, the **abstract task** of recommendation is applicable to any situation where you have two large sets of things and a particular relation between them.

The relation can be binary (it holds or it doesn't) or it can come with a numeric value that indicates the *extent* to which it holds. This could even be negative for when the relation doesn't hold.

Often, one side of the relation is a set of users and another is a set of items, but this need not always be the case. For instance, is you have a large collection of ingredients and a large collection of recipes, in which the ingredients occur, you could model this as a "recommendation" task. The resulting prediction may help to give ideas for which ingredient/recipe combinations would work well together.

The key property of the abstract taskl is that in principle, you have no information, no features, of the two types of objects beyond which ones are linked to each other. Or, if you do have some features as well, you consider it secondary information, and you want to base your predictions primarily on the property linking the two classes of objects.



the abstract task of recommendation

Given:

- a set of items of type a
  for instance: users
- a set of items of type b
  for instance: movies
- a relation linking some as to some bs
  for instance: ratings
  maybe unlabeled or labeled with a number or class

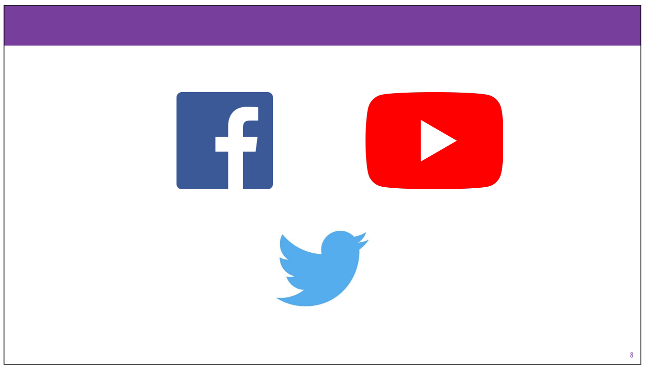
predict new links between as and bs

Ada
E.T.
likes
likes/dislikes
rates -3.141

7

The edges we predict may be unlabeled, in which case, we should simply predict whether or not a link exists, or they be be labeled. They can be labeled with classes, for instance if users can like or dislike a movie, or they can be labeled with a number, for instance with a numeric rating given to the movie.

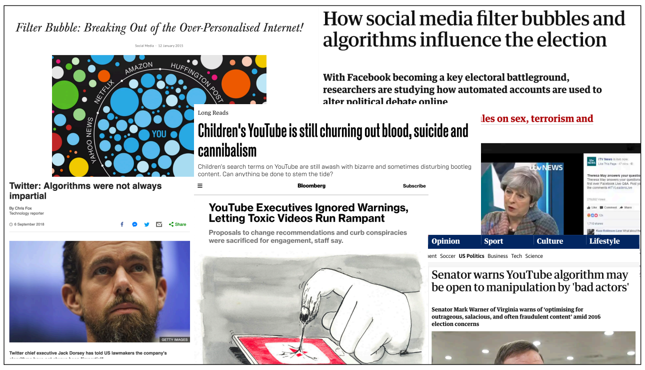
If we have something like a five star rating, it's up to us whether we prefer to model it as a relation labeled with a class or a relation labeled with a number.



8

Recommendation is probably the most widely deployed machine learning method.

In fact, in many social media platforms, recommendation is the primary means of navigation. When you load your facebook feed, your twitter timeline or your youtube homepage, the main content you see, is based on recommendation. You see the items in their database, that the algorithms think you're going to like (or at least engage with), based on your past behavior.



Filter Bubble: Breaking Out of the Over-Personalised Internet!

Social Media - 12 January 2015

How social media filter bubbles and algorithms influence the election

With Facebook becoming a key electoral battleground, researchers are studying how automated accounts are used to alter political debate online

Long Reads

Children's YouTube is still churning out blood, suicide and cannibalism

Children's search terms on YouTube are still awash with bizarre and sometimes disturbing bootleg content. Can anything be done to stem the tide?

les on sex, terrorism and

Twitter: Algorithms were not always impartial

By Chris Fox
Technology reporter

6 September 2018

Twitter chief executive Jack Dorsey has told US lawmakers the company's

Bloomberg Subscribe

YouTube Executives Ignored Warnings, Letting Toxic Videos Run Rampant

Proposals to change recommendations and curb conspiracies were sacrificed for engagement, staff say.

Opinion Sport Culture Lifestyle

ent Soccer US Politics Business Tech Science

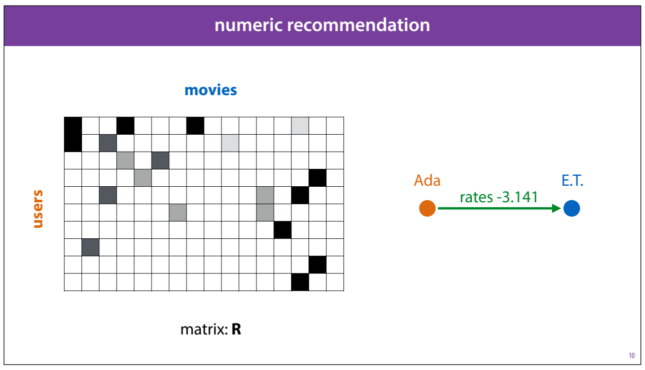
Senator warns YouTube algorithm may be open to manipulation by 'bad actors'

Senator Mark Warner of Virginia warns of 'optimising for outrageous, salacious, and often fraudulent content' amid 2016 election concerns

In fact recommendation algorithms are now so prevalent, that they are becoming a central component in the fabric of society. For a large proportion of the population, ofr instance, recommendation algorithms decide which news stories they see, and which analysis of those stories they're exposed too.

The consequences are difficult to oversee, and many issues have been discussed over the past few years. Filter bubbles may shield people from encountering different viewpoints. Optimizing algrithms for engagement may drive people towards more extreme viewpoints. And all this put together may even make the process of democracy more easy to manipulate.

In other words, it is not entirely clear at the moment whether recommendation algorithms are a force for good, or something that has grown too big for us to entirely oversee the consequences of. Either way, it pays to understand exactly how they work.



numeric recommendation

matrix: **R**

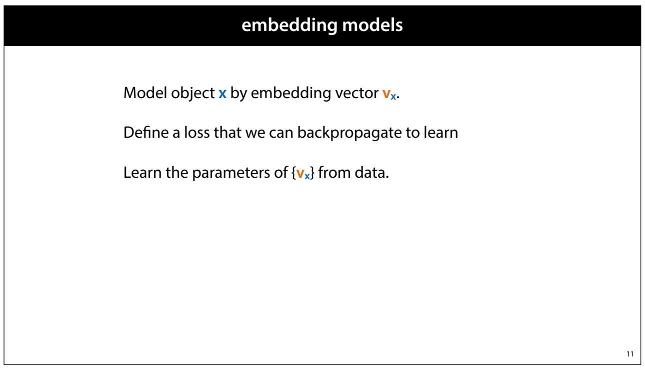
In the rest of the lecture, we'll keep to the movie recommendation use case, to keep things concrete, but verything we say can easily be adapted to other instances of the abstract recommendation task.

We'll start with the case where we have **numeric ratings**, which may be negative if a user dislikes a movie and positive if they like it. Surprisingly, this is the easiest setting to handle. We'll see later how to extend this to non-negative ratings, to class-labeled ratings and to unlabeled ratings.

We can view the space of all possible ratings as a matrix with the users along one axis, and the movies along another.

For some user/movie combinations we have a rating. Most of the matrix is empty, and these are the values that we want to predict.

The problem, as we said before, is that we have no representation for the users or for the movies. The only thing we have is two big sets of "atomic" objects, when we'd like to know when two users or two movies are similar.



embedding models

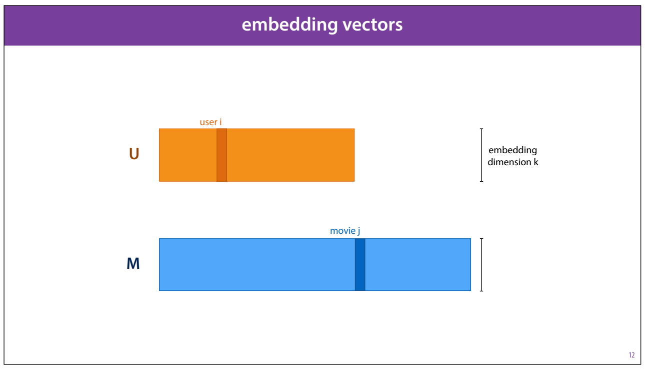
Model object $\mathbf{x}$ by embedding vector $\mathbf{v}_\mathbf{x}$.

Define a loss that we can backpropagate to learn

Learn the parameters of $\{\mathbf{v}_\mathbf{x}\}$ from data.

We've seen this problem before, in the **word embedding** problem. There, each word was an atomic object. What we did was represent each word by its own vector, and then *learn* the values of the vectors to perform some downstream task.

embedding vectors

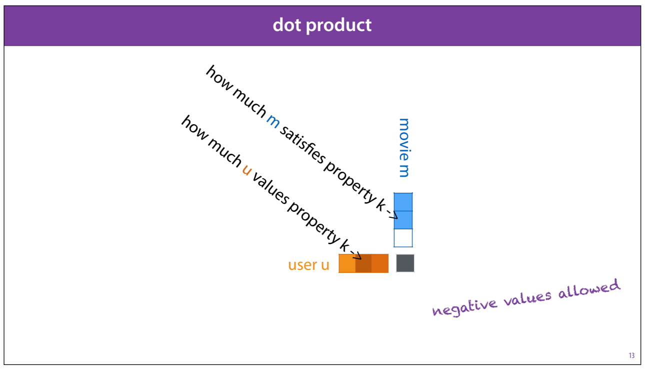


We'll apply the same trick here. We assign a vector of initially random numbers to each user and to each movie, and we will optimize the contents of these vectors to give us good representations. The number of elements k in each vector is a hyperparameter that we can set freely, but we must use the same k for both the user and the movie embeddings.

We arrange the embeddings into two matrices U and M. These are the parameters of our model.

To see how to learn these values, let's imagine first what we might do if we could set them by hand. In other words, how might we solve the problem if we could craft feature vectors for each movie and each

dot product

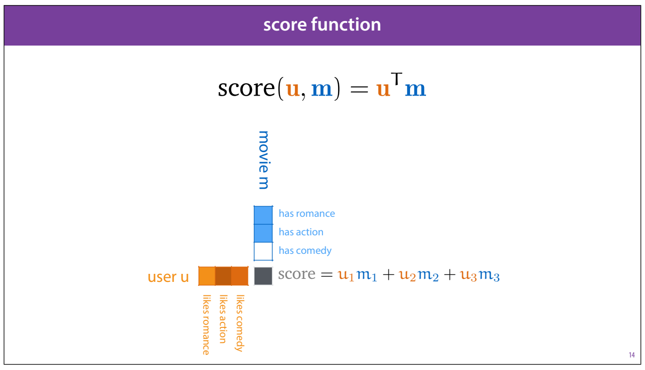


In that case, we can image setting the values by hand to represent various aspects for the users and for the movies that *match each other*. We might encode, for instance in one feature how much a user likes romance. We can make this negative for a strong dislike of romance and positive for a strong affinity of romance.

We could then then encode in the corresponding movie feature, how much romance the movie contains.

score function

$$\text{score}(\mathbf{u}, \mathbf{m}) = \mathbf{u}^T\mathbf{m}$$

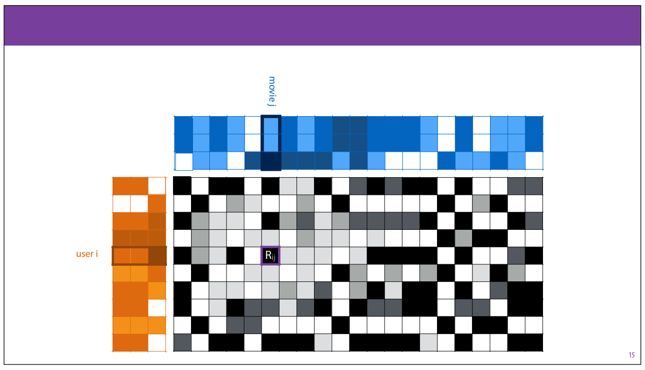


Based on these representations, we need to come up with a **score function**. Some function that takes the two representations and outputs a high positive number if the user is well-matched to the movie, a large negative number if the user will probably dislike the movie, and a number near zero if the user will be ambivalent about the movie.

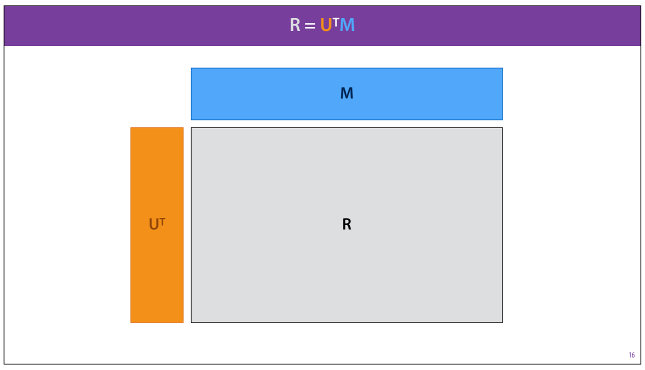
There are a few options, but a particularly simple one is the **dot product** between the user embedding and the movie embedding. This neatly expresses how much of a match the two are: if the user loves romance and the movie contains loads of it, the romance term in the sum becomes very big. The same if both values are negative (the user hates romance and the movie is very unromantic). for mismatches, the term becomes negative and the score is brought down.

A second effect is one of **magnitude**. If the user is ambivalent to romance (i.e. the romance feature is zero), that term doesn't count towards the total (and for small values, the term contributes a little bit).

Other score functions are possible, but the dot product is by far the most popular, and we'll stick with that for the rest of the lecture.

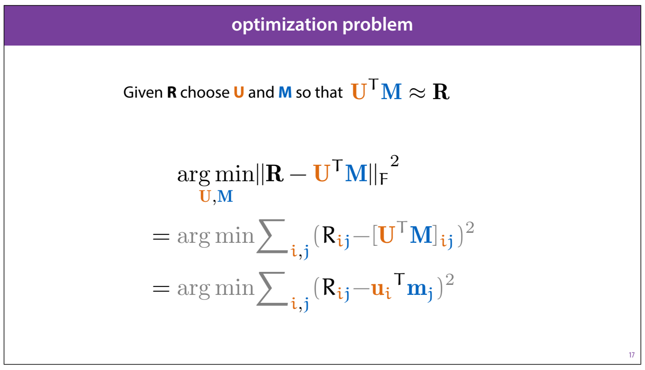


In a matrix multiplication A x B = C, each element of C contains the dot product of one row of A with one column of B. This means that multiplying U^T with **M** will gives us a matrix of rating predictions for every user/movie pair.



In other words, our aim is to take the rating matrix **R**, and to *decompose* it as the product as two factors **U** and **M**.

This is why this kind of approach to recommendation is sometimes called **matrix factorization** (or matrix decomposition). Multiplying U and M together should produce a matrix that is as close as possible to the rain matrix we have,



optimization problem

Given **R** choose **U** and **M** so that $\mathbf{U}^T\mathbf{M} \approx \mathbf{R}$

$$\underset{\mathbf{U},\mathbf{M}}{\arg\min} \|\mathbf{R} - \mathbf{U}^T\mathbf{M}\|_F^2$$
$$= \arg\min \sum_{i,j} (R_{ij} - [\mathbf{U}^T\mathbf{M}]_{ij})^2$$
$$= \arg\min \sum_{i,j} (R_{ij} - \mathbf{u}_i{}^T\mathbf{m}_j)^2$$

17

So our problem is that for a given incomplete matrix **R** of ratings, we want to find two smaller matrices **U** and **M** that multiply together into a rating matrix that is somehow close to **R**.

To turn this into an optimization objective, we need to define how to measure how close together to matrices are. The simplest option is to measure the **Frobenius norm** of the difference between the two matrices.

This sounds complicated, but it's just the same as the vector norm, but applied to matrices: we sum the squares of the elements of the matrix together and take the square root of the sum.

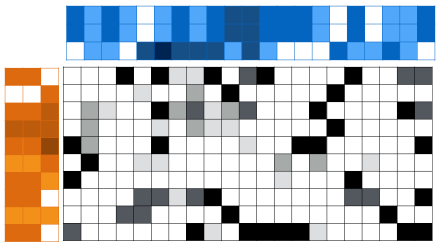
Minimizing the square of this value, is just minimizing the sum of the squared differences between the true rating matrix and our predictions. In other words, we compute **predictions** by taking the dot product of the user embedding and the movie embedding, we compute **the error** of our prediction by subtracting this from the true rating, and we minimize the sum of squared errors.

problem

We have *a lot* of missing values in **R**

One problem is that **R** is not complete. For most user/movie pairs, we don't know the rating (if we did, we wouldn't need a recommender system.

missing values

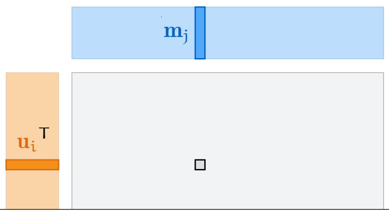


The matrix **R** is actually an *incomplete* matrix. We often fill in the unknown ratings with zeroes, but they are really *unknown* values.

If we compute the squared errors for the whole matrix, we are essentially telling our model to predict a zero rating for all of these unknown values (when actually the true ratings here may be very high or very low.

optimise only for known ratings

$$\underset{\mathbf{U},\mathbf{M}}{\arg\min} \sum_{i,j \in R_{\text{known}}} \left(R_{ij} - \mathbf{u}_i{}^T \mathbf{m}_j\right)^2$$



The solution is simple: we define the loss only for the known ratings.

This is straightforward to do if we have both positive and negative ratings, for instance likes and dislikes. We just compute the squared errors *only* over the known values of the matrix, eliminating other terms from the sum.

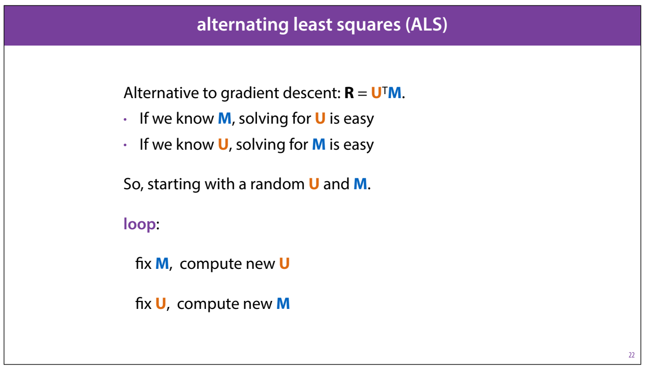
finding U and M

two options:

- alternating optimization
- gradient descent

So, now that we have our optimization objective, how do we work out good values for our embedding vectors?

The obvious choice is gradient descent. This is probably the most versatile and scalable option, but there is an alternative: alternating optimization.



alternating least squares (ALS)

Alternative to gradient descent: $\mathbf{R} = \mathbf{U}^\mathsf{T}\mathbf{M}$.

- If we know **M**, solving for **U** is easy
- If we know **U**, solving for **M** is easy

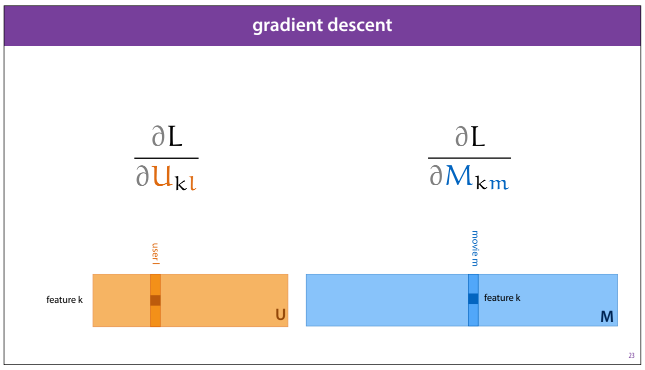
So, starting with a random **U** and **M**.

```
loop:
    fix M,  compute new U
    fix U,  compute new M
```

22

We won't dig into it deeply, but here is the basic principle. The equation **R** = **U**ᵀ**M** is is a simple linear equation with two unknowns. It's easy to solve analytically if we had one unknown (using basic linear algebra methods).

ALS has some computational benefits for small datasets, but in practice, gradient descent seems to be more flexible, for instance in dealing with missing values, different loss functions and in and adding various regularizers.



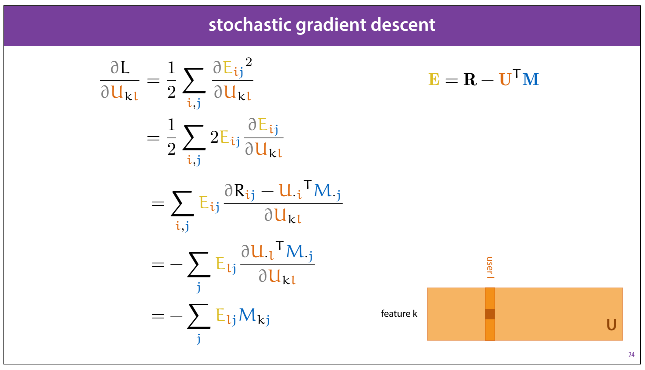
gradient descent

$$\frac{\partial L}{\partial U_{kl}} \qquad\qquad \frac{\partial L}{\partial M_{km}}$$

23

The simplest way to apply gradient descent is to implement recommendation in an automatic differentation system. If we do that, we can just define U and M as parameters, compute our loss and backpropagate. However, it's instructive to work out the gradients for the squared error loss by hand. They're not that complex, and they give us some insight into exactly how the algorithm updates the embedding values.

To do this, these are the gradients we need to work out. The gradient of the loss L for the k-th embedding value of the embedding of user l, and similarly for the movie embeddings.



stochastic gradient descent

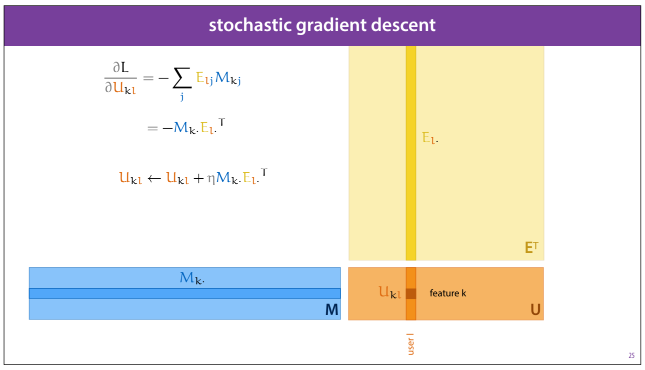
$$\begin{aligned}\frac{\partial L}{\partial U_{kl}} &= \frac{1}{2}\sum_{i,j}\frac{\partial {E_{ij}}^2}{\partial U_{kl}} \\ &= \frac{1}{2}\sum_{i,j}2E_{ij}\frac{\partial E_{ij}}{\partial U_{kl}} \\ &= \sum_{i,j}E_{ij}\frac{\partial R_{ij} - {U_{.i}}^\mathsf{T}M_{.j}}{\partial U_{kl}} \\ &= -\sum_{j}E_{lj}\frac{\partial {U_{.l}}^\mathsf{T}M_{.j}}{\partial U_{kl}} \\ &= -\sum_{j}E_{lj}M_{kj}\end{aligned}$$

$$\mathbf{E} = \mathbf{R} - \mathbf{U}^\mathsf{T}\mathbf{M}$$

24

To apply gradient descent, we need to work out the gradients for our parameters: the embeddings of the users and the embeddings of the movies.

We could implement the loss computation in a deep learning system, and let backpropagation take care of this, but we can actually work out the gradients ourselves. It's instructive to do that here, and to see how gradient descent updates the parameters.



stochastic gradient descent

$$\begin{aligned}\frac{\partial L}{\partial U_{kl}} &= -\sum_{j}E_{lj}M_{kj} \\ &= -M_{k.}{E_{l.}}^\mathsf{T}\end{aligned}$$

$$U_{kl} \leftarrow U_{kl} + \eta M_{k.}{E_{l.}}^\mathsf{T}$$

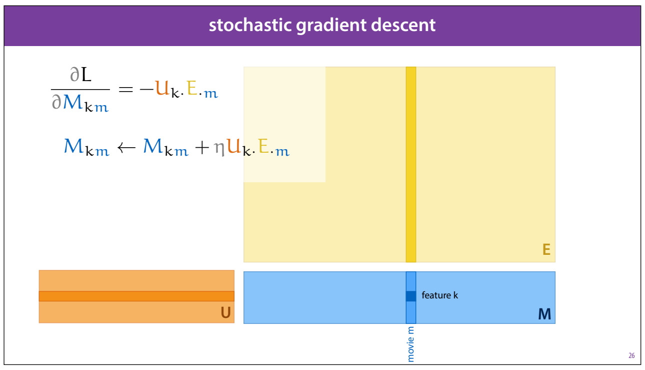
25

We update the k-th value of the embedding for user l, by computing the error vector for user l *over all movies*, and taking the dot product with the k-th feature over all movies.

Imagine that the k-th value of the user and movie embeddings represents how romantic the user and movie are respectively. Now imagine that we had a movie that we think is very romantic and a user that we think is very romantic, that is, the both have high values for the k-th value in their embedding. Since the embeddings match well, we end up giving a high rating.

Now imagine that the actual rating was much lower, so that we end up with a negative error: element E_{lm} is a large negative number. The update rule tells us what this means. The movie's k-th element was high, and we're taking that as a constant at the moment. Therefore, we can only assume that the large error was due to the user. We update the user's k-th value by the error multiplied by the movie's k-th value, subtracting a large value.

In short, assuming that both the movie and the user were romantic gave us a large error, and we are treting the movie as a constant, so we conclude that the user must be less romantic than we throught.



stochastic gradient descent

$$\frac{\partial L}{\partial M_{km}} = -U_{k\cdot}E_{\cdot m}$$

$$M_{km} \leftarrow M_{km} + \eta U_{k\cdot}E_{\cdot m}$$

When we look at the update for the movie, wesee the opposite. If the same thing happened: the user and the movie both have a high k-th value, we assume both are romantic, we give the pair a high rating and get a negative error, then we end up making the movie less romantic, since we are treating the romanticness of the user as a given.

In practice, of course, we apply both update rules. So both the move and the user end up getting a little less romantic.

source: Matrix Factorization Techniques for Recommender Systems, Yehuda Koren et al (2009).

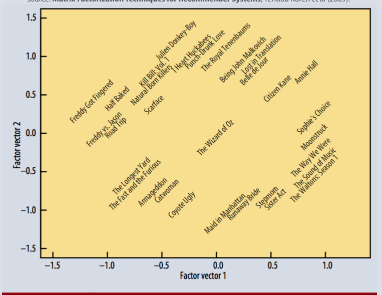
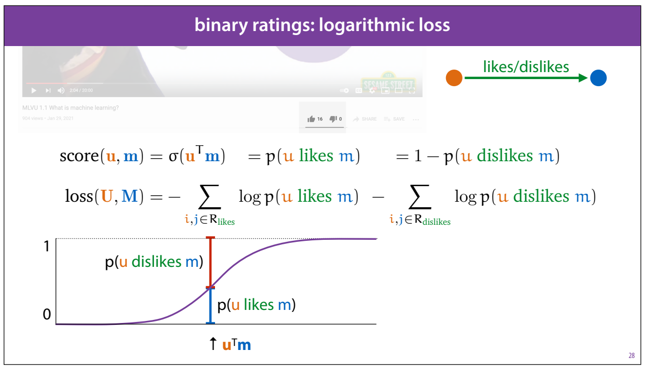


Figure 3. The first two vectors from a matrix decomposition of the Netflix Prize data. Selected movies are placed at the appropriate spot based on their factor vectors in two dimensions. The plot reveals distinct genres, including clusters of movies with strong female leads, fraternity humor, and quirky independent films.

source: **Matrix Factorization Techniques for Recommender Systems**, Yehuda Koren et al (2009).

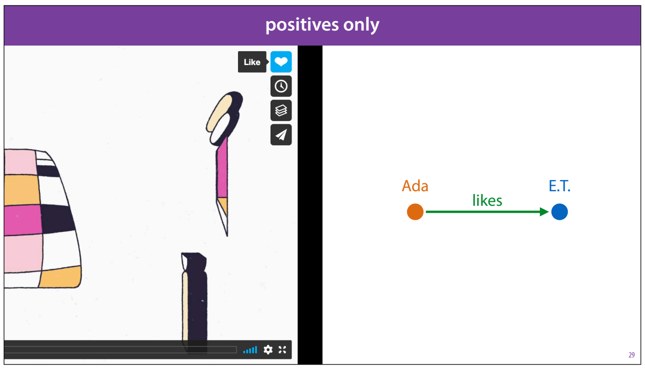


binary ratings: logarithmic loss

$$\text{score}(\mathbf{u}, \mathbf{m}) = \sigma(\mathbf{u}^T\mathbf{m}) \quad = p(u \text{ likes } m) \qquad = 1 - p(u \text{ dislikes } m)$$

$$\text{loss}(\mathbf{U}, \mathbf{M}) = -\sum_{i,j\in R_{\text{likes}}} \log p(u \text{ likes } m) \;-\; \sum_{i,j\in R_{\text{dislikes}}} \log p(u \text{ dislikes } m)$$

If the rating system is binary, like the like/dislike on Youtube and Netflix, then the scores for each user/item pair are best understood as classes. We can turn our dot product score into a binary class by applying a sigmoid to the dot product and applying a logarithmic loss. We interpret the value after the sigmoid as the probabiltiy that the user will like the movie, and 1 minus that value as the probability that the user will dislike the movie, and we take the negative logarithm of the probability of the correct option as our loss.



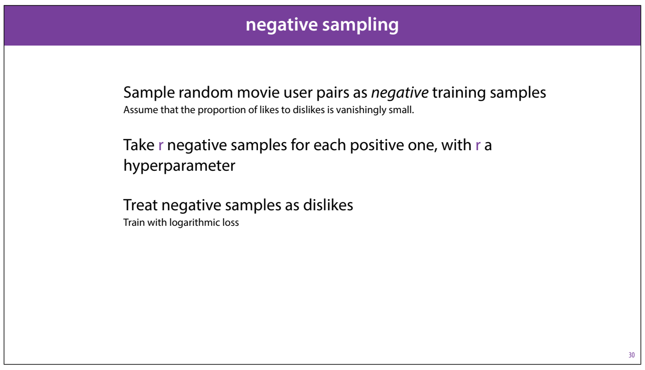
positives only

In many recommender systems, we only get positive ratings. You can “like” something, but you can’t dislike it or assign a number.

The benefit of such rating systems is that users are much more likely to give ratings. First, because it’s less work, and second because it has a direct benefit for the user. They’re no just doing it to improve their recommendations (which they may not care about), they are effectively bookmarking the things they like, so that they can easily find them again. Thus you’re likely to get many more ratings if you build your system this way.

The downside is that the modeling task is much more complicated. It’s like a classification task where the only labels you get are positive and *unknown*. For the unknowns, you don’t know how many positives there are, and how many negatives.

If we just optimize the score function to be as big as possible for the known likes, then there’s nothing stopping the system from making the ratings as high as it can for all user/movie pairs.



negative sampling

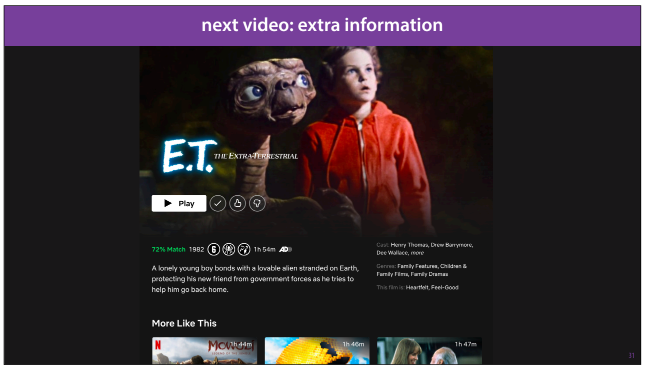
Sample random movie user pairs as *negative* training samples
Assume that the proportion of likes to dislikes is vanishingly small.

Take r negative samples for each positive one, with r a hyperparameter

Treat negative samples as dislikes
Train with logarithmic loss

A common and very effective trick is to sample random pairs of users and items and *assume* that these are negatives. Usually the proportion of positive user/movie pairs is vanishingly small compared to the proportion of pairs that are negative, or pairs for which the users are ambivalent, so if we sample a random pair, we can be almost certain that the user won’t like the movie.

With these negative samples in hand, we can treat the problem as a binary classification problem and re-use what we learned for class-labeled ratings.



next video: extra information

72% Match 1982 6 1h 54m AD

A lonely young boy bonds with a lovable alien stranded on Earth, protecting his new friend from government forces as he tries to help him go back home.

Cast: Henry Thomas, Drew Barrymore, Dee Wallace, *more*

Genres: Family Features, Children & Family Films, Family Dramas

This film is: Heartfelt, Feel-Good

More Like This

So far, we’ve assumed that we don’t have any information about users and movies by themselves: only the links between them. In practice, this isn’t true at all: Netflix has lots of extra information about both the users and the movies in its database. It’s just that we’ve assumed that the ratings are the most informative, so that we should start there.

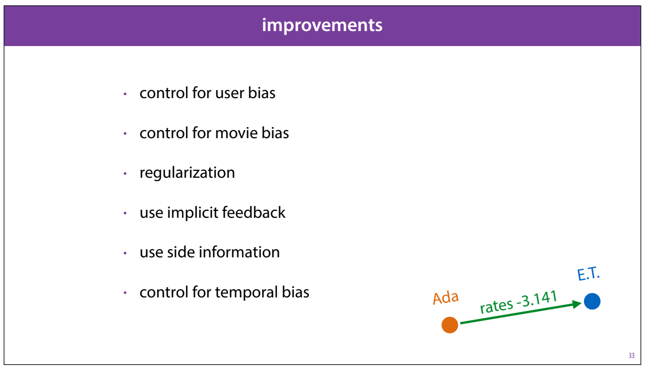
Of course, ideally, we don’t want to dismiss any information we have. In the next video, we’ll look at how we can extend a recommendation system with extra sources of information.

Embedding models

Part 2: Improving recommenders

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this video, we'll look at how we can take the basic model from the last video, and extend it to improve its performance.



improvements

- control for user bias
- control for movie bias
- regularization
- use implicit feedback
- use side information
- control for temporal bias

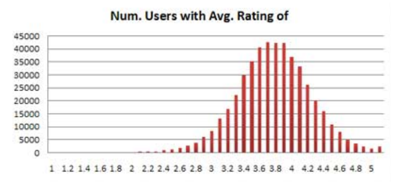
Ada
rates -3.141
E.T.

33

These are the topics we'll deal with.

Most of these tricks are based on the system that ultimately won the Netflix prize, so we'll assume that we have numeric ratings.

user bias



source: http://www.hackingnetflix.com/2006/10/netflix_prize_d.html

34

The average rating for each user is different. Some users are very positive, giving almost every movie 5 stars, and some give almost every movie less than 3 stars.

If we can explicitly model the bias of a user, it takes some of the pressure off the matrix factorisation, which then only needs to predict how much a user will deviate from their average rating for a particular movie.

movie bias

35

The same is true for movies. Some movies are universally liked, and some are universally loathed.

biases

b: generic bias

b_i: user bias

b_j: movie bias

$$\text{score}(i, j) = \mathbf{u}_i^{\mathsf{T}} \mathbf{m}_j + b_i + b_j + b$$

We model biases by a simple additive scalar (which is learned along with the embeddings): one for each user, one for each movie, and one general bias over all ratings.

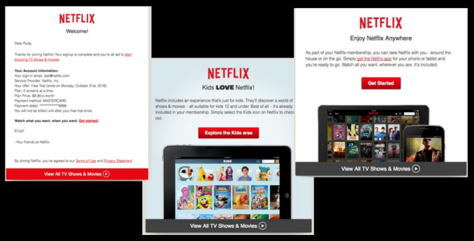
We can think of these parameters as taking some of the weight off the embeddings. If user i is very positive, and we didn't have bias terms, we'd need to set their embedding so that it's postive for all movies. With the bias term, the dot product just needs to model the distance to the user's average rating.

L2 regulariser

$$\underset{\mathbf{U},\mathbf{M}}{\arg\min}\ \text{loss}(\mathbf{U}, \mathbf{M}) \quad + \lambda_u \|\mathbf{U}\|_F{}^2 \quad + \lambda_m \|\mathbf{M}\|_F{}^2$$

The more weights we add for the users and the movies, the more likely our model is to overfit. If this is a danger, then it may help to regularize a little. We can to this by a simple regularization term over the parameters.

cold start problem



source: https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-how-netflix-achieves-email-delivery-at-global-scale-with-amazon-ses-mbl204

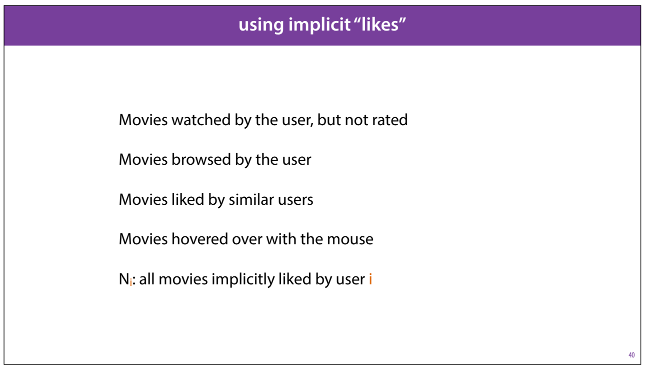
One big problem in recommender systems is the **cold start problem**. When a new user joins Netflix, or a new movie is added to the database, we have no ratings for them, so the matrix factorization has nothing to build an embedding on.

I this case we have to rely on implicit feedback and side information.

implicit feedback

Proxies for possible ratings:

- Page views
- Wishlist
- Record cursor movement



using implicit "likes"

Movies watched by the user, but not rated

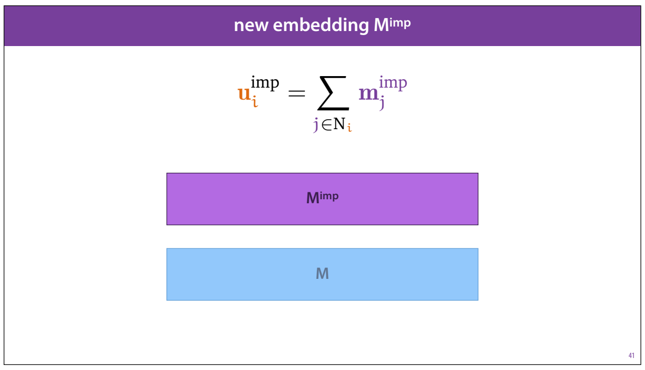
Movies browsed by the user

Movies liked by similar users

Movies hovered over with the mouse

N_i: all movies implicitly liked by user i

All of these are useful information, but we don't want to treat them the same as our regular likes. ultimately, they're much less reliable and they should be interpreted differently.



new embedding M^{imp}

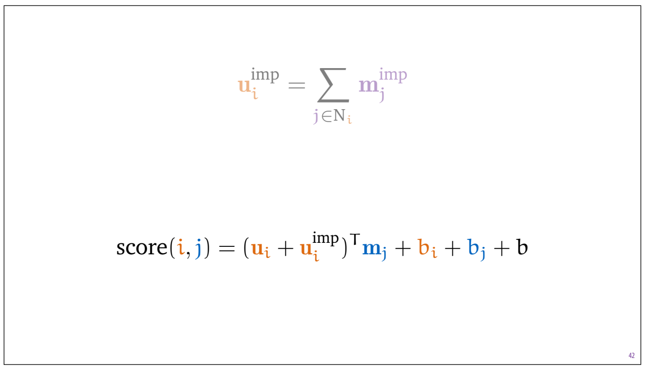
$$\mathbf{u}_i^{imp} = \sum_{j \in N_i} \mathbf{m}_j^{imp}$$

There are different ways of handling this problem, but this is the method used in the system that won the Netflix prize.

We add a second matrix of movie embeddings M^{imp}, and then compute a new user embedding which is the sum of the x-embeddings of all the movies user x has implicitly "liked". That is, we simply sum up the embeddings of all the movies the user has in some way been associated with. This sum functions as a second embedding for the user i.

Ni is the set of movies with which user i is associated through implicit information.

Note that there is a slightly counter-intuitive step here: we are learning *movie* embeddings, but their only function is to become a representation of the *user*.



$$\mathbf{u}_i^{imp} = \sum_{j \in N_i} \mathbf{m}_j^{imp}$$

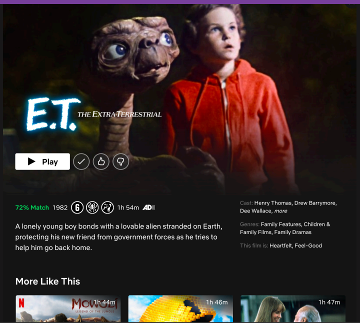
$$\text{score}(i, j) = (\mathbf{u}_i + \mathbf{u}_i^{imp})^T \mathbf{m}_j + b_i + b_j + b$$

We then add the implicit-feedback embedding to the existing one before computing the dot product.

To understand what's happening, let's look at the edge cases. If the implicit associations don't help at all, all embedding vectors m^{imp} will simply go to zero. If they help for some movies but not for others, then only the vectors of some movies will become non-zero.

The fact that the movie recommendations are used for

extra information


E.T. THE EXTRA-TERRESTRIAL

72% Match 1982 1h 54m

A lonely young boy bonds with a lovable alien stranded on Earth, protecting his new friend from government forces as he tries to help him go back home.

Cast: Henry Thomas, Drew Barrymore, Dee Wallace, *more*

Genres: Family Features, Children & Family Films, Family Dramas

This film is: Heartfelt, Feel-Good

More Like This

43

As we noted in the last video, we do usually have features for the movies and the users as well, it's just that the ratings are more predictive. Can we use the features for the users and movies to boost our performance, and to help with the cold start problem?

side information (features)

about movies	about users
length	country
genre	language
actors, director	OS
synopsis	login times
awards	bio
	social media profile
	connection to other users

44

Here's some of the information we may have for users and movies.

This is essentially a big instance/feature matrix like we've seen already in the classic setting: one for the movies and one for the users.

The challenge is to integrate this with the ratings, so that we can extend the relatively sparse information we get from those by generalising over the sets of users and movies.

using side information

User features: age, login, browser resolution, social media
Binary features only, for simplicity

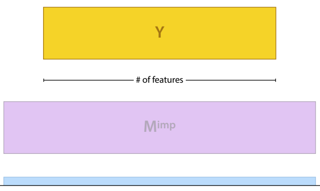
A_i: all features that apply to user u_i

45

To simplify things, we'll assume all features are *binary categories*: the feature applies or it doesn't.

new embedding Y

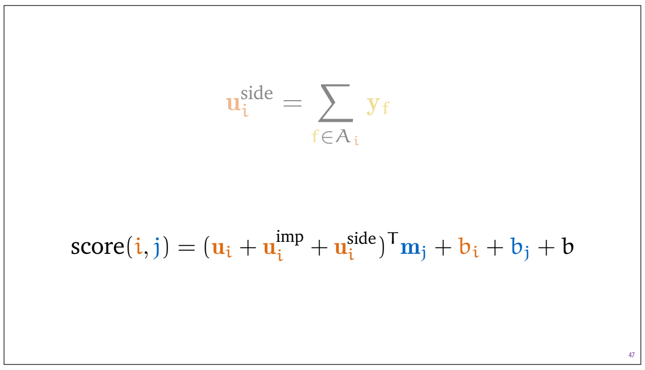
$$\mathbf{u}_i^{\text{side}} = \sum_{f \in A_i} \mathbf{y}_f$$



46

We follow the same logic as we did before, for the implicit feedback, and add another matrix of embedding vector: one embedding for each feature that can apply to a user. We sum up all the features that apply to user i and get another representation for the user.

We then assign each feature an embedding, and sum over all features that apply to the user, creating a third user embedding.



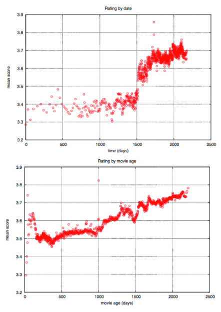
$$\mathbf{u}_i^{\text{side}} = \sum_{f \in A_i} \mathbf{y}_f$$

$$\text{score}(i, j) = (\mathbf{u}_i + \mathbf{u}_i^{\text{imp}} + \mathbf{u}_i^{\text{side}})^T \mathbf{m}_j + b_i + b_j + b$$

47

We add it to the sum inside the dot product. We could do the same for the movie information, but we will take that as read, to keep the score simple

time

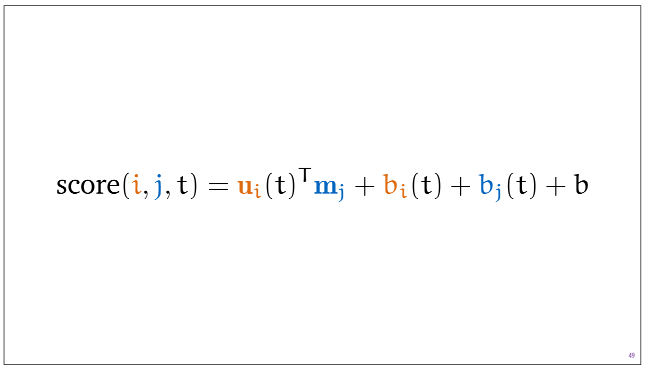


source: Collaborative Filtering with Temporal Dynamics, Yehuda Koren

48

The Netflix data is not stable over time. It covers about 7 years, and in that time many things have changes. The most radical change comes about four years in, when Netflix changed the meaning of the ratings in words (these appeared in mouseover when you hovered over the ratings). Specifically, they changed the one-star rating from “I didn’t like it” to “I hated it”. Since people are less likely to say that they hate things, the average ratings increased.

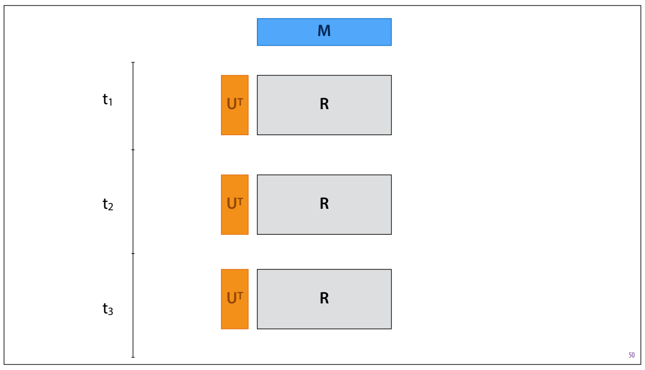
Similarly, if you look at how old a movie is, you see a positive relation to the average rating. Generally, people who watch a really old movie will likely do so because they know it, and want to watch. Whereas for new movies, more people are likely to be swayed by novelty and advertising. This means that new movies have a temporal bias for lower ratings.



$$\text{score}(i, j, t) = \mathbf{u}_i(t)^T \mathbf{m}_j + b_i(t) + b_j(t) + b$$

49

The solution is to make the biases, and the user embeddings **time dependent**. For the movies we make only the bias time dependent, since the properties of the movie itself stay the same. For user embeddings, we can actually make the embeddings time dependent, since user tastes may change over time.



A very practical way to do this is just to cut time into a small number of chunks and learn a separate embedding for each chunk. Note that all the matrices stay the same size. There are just fewer ratings in **R**.

The more chunks we cut time into, the better we can model the time-dependency, but the worse our individual embeddings get, since we have less and less data per chunk.

source: **Matrix Factorization Techniques for Recommender Systems**, Yehuda Koren et al (2009).

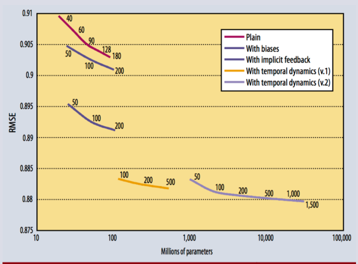


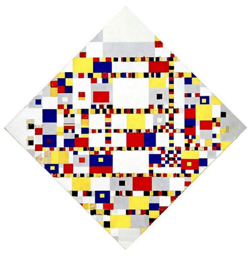
Figure 4. Matrix factorization models' accuracy. The plots show the root-mean-square error of each of four individual factor models (lower is better). Accuracy improves when the factor model's dimensionality (denoted by numbers on the charts) increases. In addition, the more refined factor models, whose descriptions involve more distinct sets of parameters, are more accurate. For comparison, the Netflix system achieves RMSE = 0.9514 on the same dataset, while the grand prize's required accuracy is RMSE = 0.8563.

Here is how the different additions to the basic matrix factorization ultimately served to reduce the RMSE to the point that won the authors the Netflix prize.

summary

When your task consists of linking one large set of things to another large set of things, based on sparse examples, and little intrinsic information, **matrix factorization** may be appropriate.

Extend your models with biases, regularizers, implicit likes, side information and temporal dynamics.

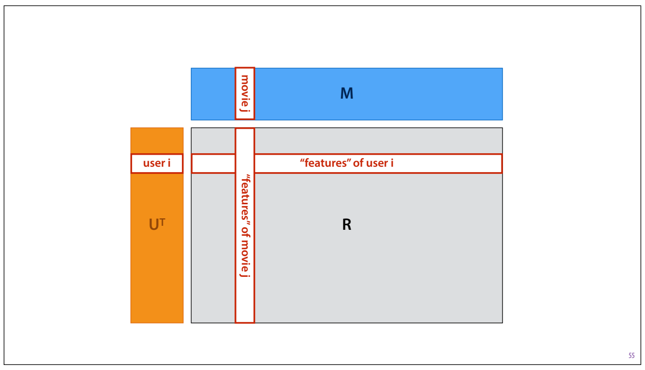


That's it for recommendation. In the last videos in the lecture we'll take a few quick looks at other places where embedding models can give us a new perspective, and we'll finish up with some general notes on how to validate embedding models

Embedding models

Part 3: PCA revisited

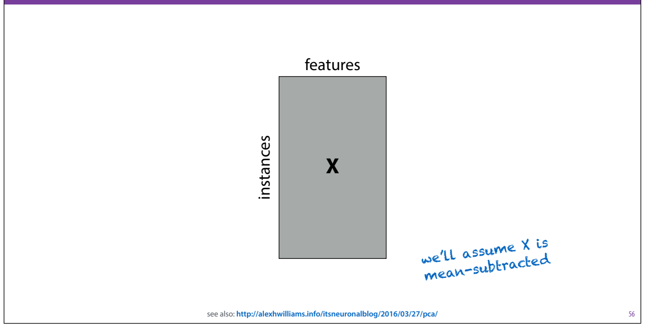
Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam



In the previous video, we saw that we could get embeddings by thinking of our data as a big matrix, and decomposing it into matrics of embeddings.

We can think of this as two traditional data matrices in one: if we consider the users as instances, then the movies are a big set of binary features. If we consider the movies as instances, then which users they are liked by are their features.

classic ML: factoring the data matrix

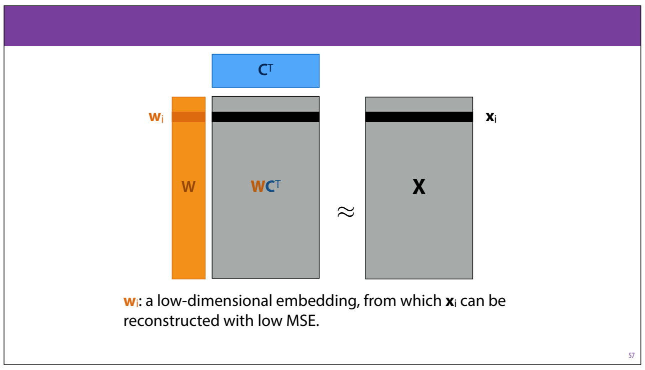


see also: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

In the classical machine learning setting, our data can also be seen as a matrix (usually with an instance per row, and a feature per column.

What would happen if we apply matrix factorization to this matrix?

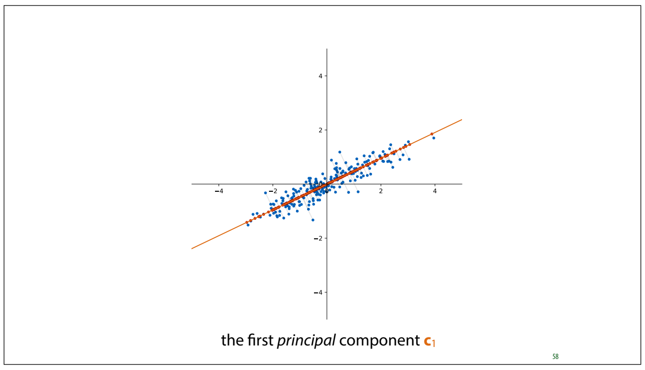
NB: In the following we'll assume that the data have been mean-subtracted (the mean over all rows has been subtracted from each row).



$\mathbf{w}_i$: a low-dimensional embedding, from which $\mathbf{x}_i$ can be reconstructed with low MSE.

If we apply the same principle as we did with the recommender system, we are looking for two matrices, W and C: the first containing "embeddings" for our instances and the second "embeddings" for our features, such that their dot product reconstructs, as much as possible, the value of a particular feature for a particular matrix.

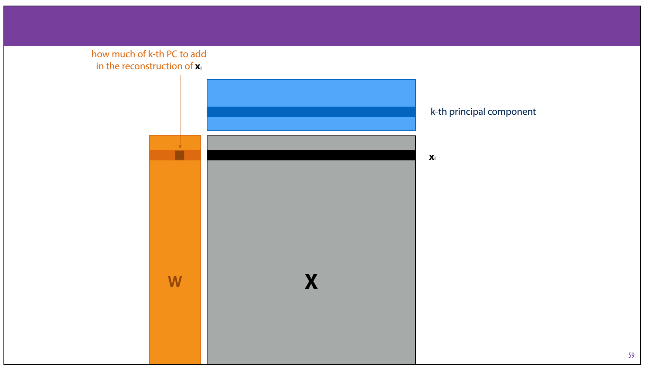
If we do this succesfully,we get a dimensionality reduction. One based on a linear transformation.



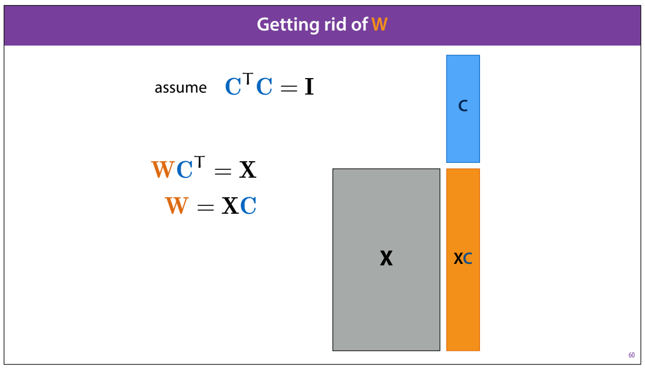
the first *principal* component $\mathbf{c}_1$

58

Our first dimensionality reduction method, PCA, was also based on a linear transformation.


how much of k-th PC to add in the reconstruction of $\mathbf{x}_i$
k-th principal component
$\mathbf{x}_i$
W
X

59



Getting rid of W

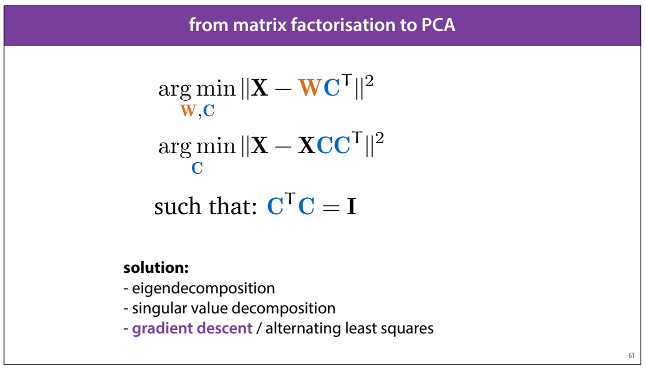
assume $\mathbf{C}^\mathsf{T}\mathbf{C} = \mathbf{I}$

$$\mathbf{W}\mathbf{C}^\mathsf{T} = \mathbf{X}$$
$$\mathbf{W} = \mathbf{X}\mathbf{C}$$

60

In PCA, we assume that the principal components were unit vectors and orthogonal to each other. We can do the

by assuming that the columns of C are linearly independent. In this case, we can rewrite W in terms of C,and reduce the parameters of the model to just the “feature embeddings”.



from matrix factorisation to PCA

$$\underset{\mathbf{W},\mathbf{C}}{\arg\min} \|\mathbf{X} - \mathbf{W}\mathbf{C}^\mathsf{T}\|^2$$

$$\underset{\mathbf{C}}{\arg\min} \|\mathbf{X} - \mathbf{X}\mathbf{C}\mathbf{C}^\mathsf{T}\|^2$$

such that: $\mathbf{C}^\mathsf{T}\mathbf{C} = \mathbf{I}$

solution:
- eigendecomposition
- singular value decomposition
- gradient descent / alternating least squares

61

This gives us a constrained optimisation problem that is very close to PCA. It’s not entirely equivalent, but PCA is one of the solutions to this problem.

See: **peterbloem.nl/blog/pca-2** for the full story.

incomplete PCA

$$\arg\min_{\mathbf{C}} \sum_{i,j \in \text{known}} (X_{ij} - [\mathbf{W}^T\mathbf{C}]_{ij})^2$$

Dimensionality reduction/data completion

62

This perspective e allows us to modify the PCA objective with the tricks we've seen in the recommender setting.

For instance, if our data has **missing values**, we can focus the optimization only on the known values, giving us a mixture of dimensionality reduction and data completion.

We can then learn on the low-dimensional representations, or reconstruct the data to give us imputations for the missing data

L2 regulariser

$$\arg\min_{\mathbf{W},\mathbf{C}} \|\mathbf{X} - \mathbf{W}\mathbf{C}^T\|_F^{\ 2} + \gamma_1 \sum_i \|\mathbf{w}_i\|_2^{\ 2} + \gamma_2 \sum_j \|\mathbf{c}_j\|_2^{\ 2}$$

63

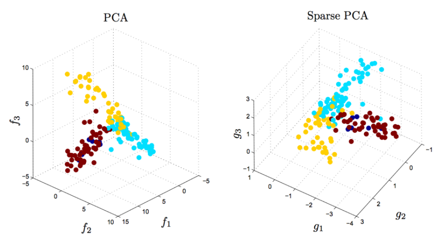
We can also add a regularizer to constrain the complexity of our embeddings.

L1 regulariser: sparse PCA

$$\arg\min_{\mathbf{W},\mathbf{C}} \|\mathbf{X} - \mathbf{W}\mathbf{C}^T\|_F^{\ 2} + \gamma_1 \sum_i \|\mathbf{w}_i\|_1^{\ 2} + \gamma_2 \sum_j \|\mathbf{c}_j\|_1^{\ 2}$$

64

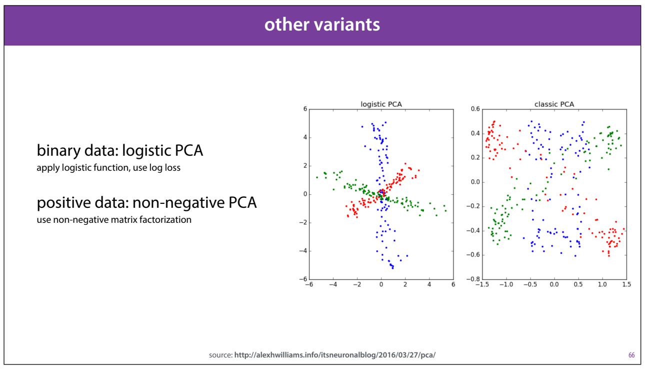
An L1 regularizer, as we know, promotes sparse models: models for which parameters are exactly zero. In this case that means that our embeddings are more likely to contain zeroes, which can make it easier to interpret the results.



source: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

65

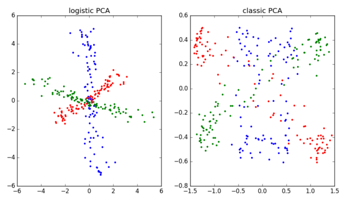
Here's an example of a dataset reduced to 3D. It's a bit difficult to see, but the points in the sparse PCA should be more axis aligned.



other variants

binary data: logistic PCA
apply logistic function, use log loss

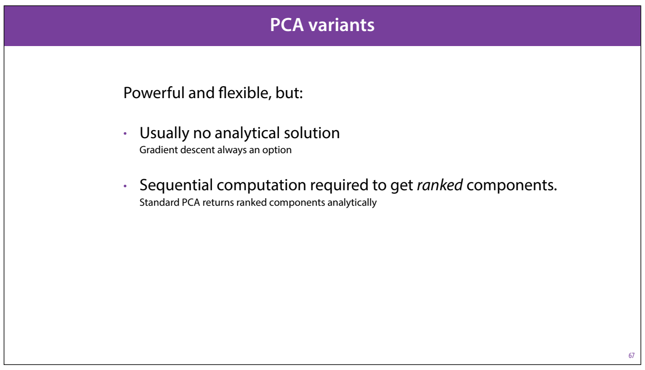
positive data: non-negative PCA
use non-negative matrix factorization



source: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

If our data has binary values, then we can reduce it with the same trick we saw before: apply a sigmoid and fit the log loss. As you can see on the right, this often gives us much better separation in the reduced dimensionality.

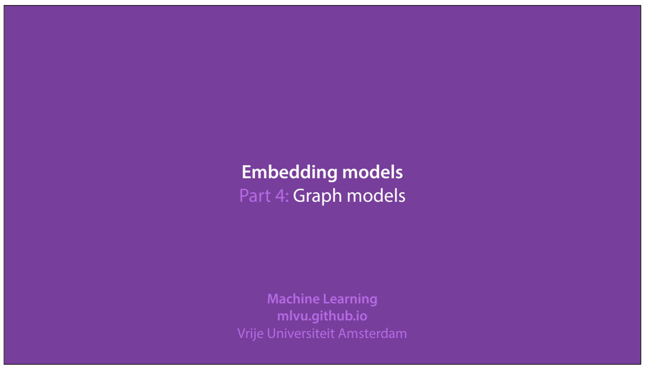
If we have binary data that is largely missing, for which we only know some of the postives, non-negative matrix factorization gives us non-negative PCA.



PCA variants

Powerful and flexible, but:

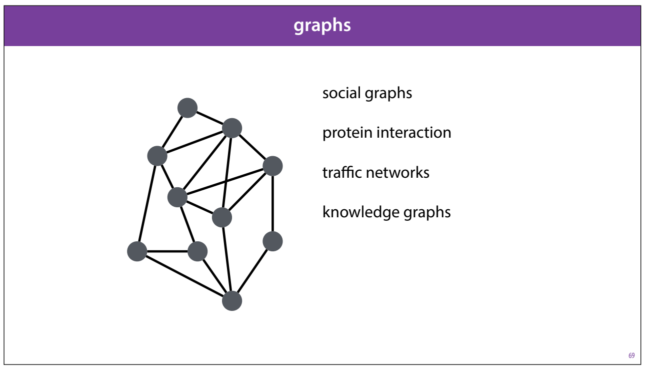
- Usually no analytical solution
  Gradient descent always an option
- Sequential computation required to get *ranked* components.
  Standard PCA returns ranked components analytically



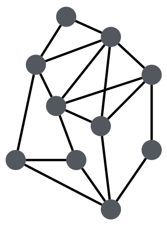
Embedding models
Part 4: Graph models

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this video, we'll look at how some of these concepts can be applied to graphs. This is a complex subject, so we'll only give a very high-level overview, without going into many details.



graphs



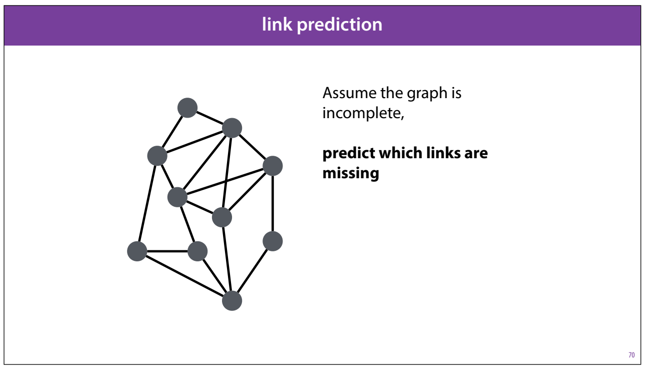
social graphs

protein interaction

traffic networks

knowledge graphs

Graphs are an even more versatile format for capturing knowledge than matrices and tensors. Many of the most interesting datasets come in the form of graphs.

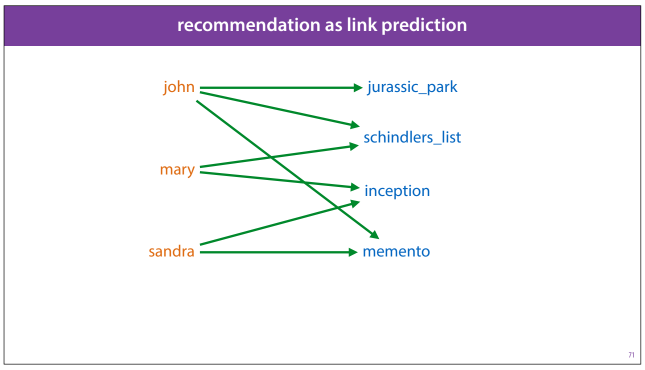


link prediction

Assume the graph is incomplete,

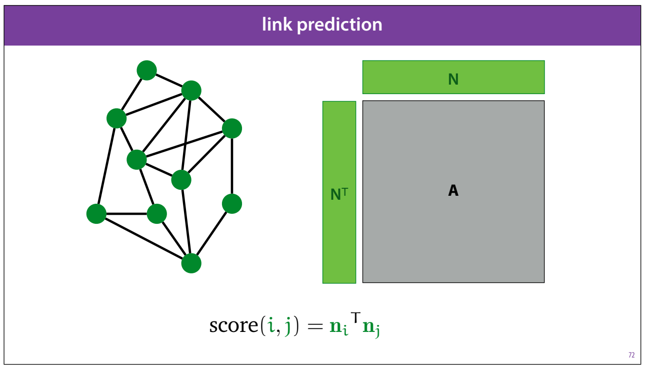
predict which links are missing

In link prediction, we assume the graph we see is incomplete (which is usually the case) and we try to predict which nodes should be linked .



recommendation as link prediction

We can see recommendation as a particular instance of the link prediction problem. Here, the graph is bipartite: we have two different *types* of nodes (users and movies), and links are always from one type to the other.



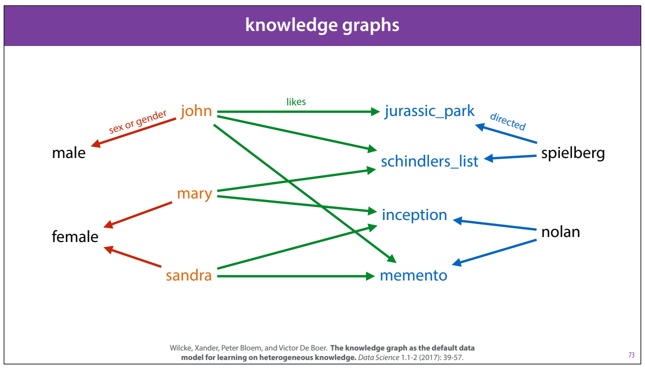
link prediction

$$\text{score}(i, j) = \mathbf{n}_i{}^T \mathbf{n}_j$$

In general link prediction, we graph may not be bipartite, so we just learn a single embedding matrix for all nodes. We can then compute a score for the likelihood of a link existing in the graph between nodes i and j with the dot product, and train the embeddings to learn the known links, and use them to predict new links.

This way we can predict which proteins might interact with each other, which people in a social network may be friends (or should be friends) and so on.

In short, we apply the principle of matrix factorization to the adjacency matrix of a graph. We can then use all the tricks from the first two videos to optimize our

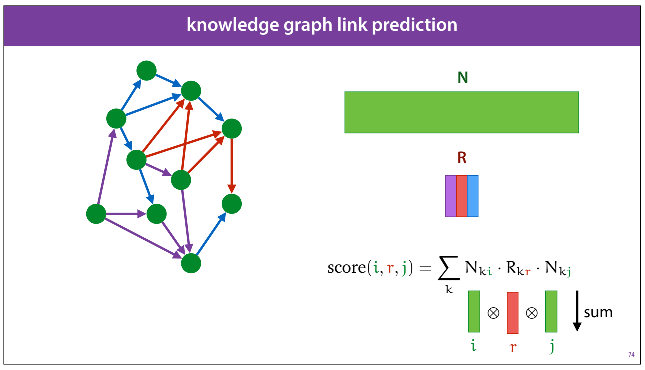


knowledge graphs

Wilcke, Xander, Peter Bloem, and Victor De Boer. **The knowledge graph as the default data model for learning on heterogeneous knowledge.** *Data Science* 1.1-2 (2017): 39-57.

Knowledge graphs are graphs where nodes represent concepts or entities, and links are labeled with a relation. It's a bit like a lot of different recommendation tasks rolled into one.

Note how the extra knowledge of different relations can potentially help our predictions of other relations. For instance, knowing that John likes Memento, and that Memento is directed by Chrisopher Nolan, may allow us to conclude that John may like Inception at well.



knowledge graph link prediction

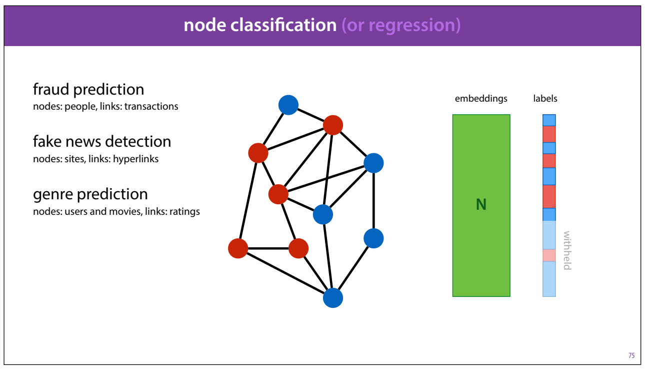
$$\text{score}(i, r, j) = \sum_k N_{ki} \cdot R_{kr} \cdot N_{kj}$$

74

There are many ways to do link prediction in knowledge graphs, but a very simple approach is to learn node embeddings as before, but to also learn a separate embedding for the different relation types.

This score function is called “distmult”, but many others exist with differing levels of complexity.

We can think of this as decomposign a 3-tensor into the product of three embedding matrices.



node classification (or regression)

fraud prediction
nodes: people, links: transactions

fake news detection
nodes: sites, links: hyperlinks

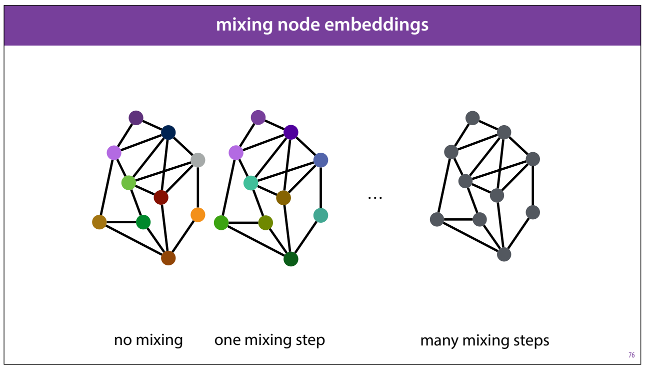
genre prediction
nodes: users and movies, links: ratings

75

Node classification is another task: for each node, we are given a label, which we should try to predict.

If we have vector representations for our nodes, we can use those in a regular classifier, but the question is, how do we get those embeddings, and how do we ensure that they capture the required information?

We can’t just assign node embeddings like before, and apply gradient descent on the classification loss. That would ignore the graph structure entirely and would train each embedding in isolation to produce a particular clasification. We wouldn’t generalize *between* nodes.



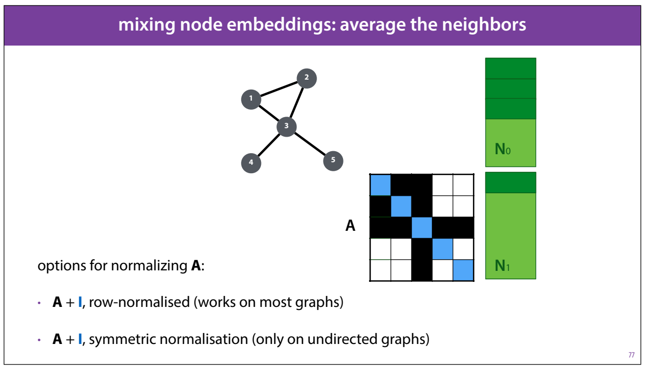
mixing node embeddings

76

The principle we will be using to learn/refine our embedding is that of **mixing embeddings**. To develop our intuition, imaging that we assign all nodes random 3D embeddings, with values between 0 and 1. For the purposes of visualization, we can then interpret these as RGB colors. We start with an entirely random color per node.

We then apply a mixing step: we replace each node color by the mixture (the average) of itself and its direct neighbors. At first, the embeddings express nothing but identity, each node has its own color. After one mixing step, the node embeddings express something about the local graph neighbourhood: a node that is close to many purple nodes will come slightly more purple itself.

After many mixing steps, all nodes have the same embedding, expressing only information about the entire graph. Somewhere in-between we find a sweet spot: where the embeddings express the node identity, but also the structure of the local graph neighborhood.



mixing node embeddings: average the neighbors

options for normalizing **A**:

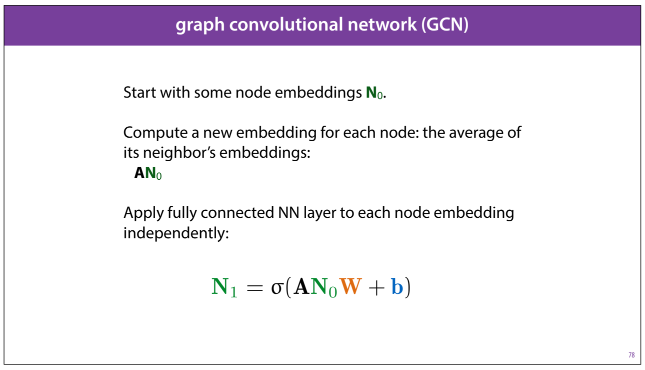
- **A** + **I**, row-normalised (works on most graphs)
- **A** + **I**, symmetric normalisation (only on undirected graphs)

The simplest way to mix node embeddings is just to make the new node embedding the sum or average of of all the embeddings of the neighbors.

We can achieve this mixing by multiplying the embedding vector by the adjacency matrix: this results in the sum of the embeddings of the neighbouring nodes. We also add self-loops for every node so that the current embedding stays part of the sum.

If we sum, the embedding will blow up. with every mixing step. In order to control for this we need to normalize the adjacency matrix. If we row-normalize, we get the average over all neighbours. We can also use a **symmetric normalization**, which leads to a slightly different type of mixing but only works on undirected graphs.

See this article for more details: **https://tkipf.github.io/graph-convolutional-networks/**



graph convolutional network (GCN)

Start with some node embeddings $\mathbf{N}_0$.

Compute a new embedding for each node: the average of its neighbor's embeddings:
$\mathbf{A}\mathbf{N}_0$

Apply fully connected NN layer to each node embedding independently:

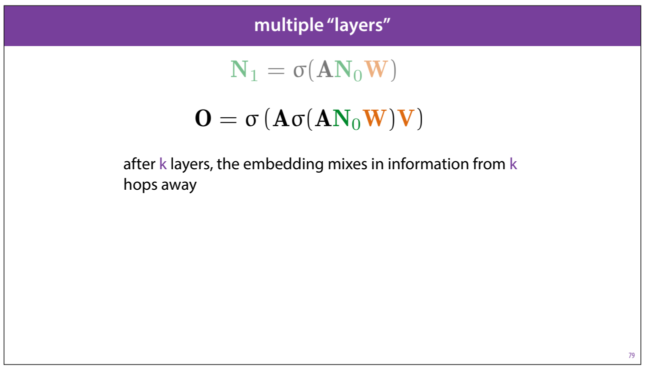
$$\mathbf{N}_1 = \sigma(\mathbf{A}\mathbf{N}_0\mathbf{W} + \mathbf{b})$$

This is the principle used in graph convolution networks. The word convolution is used because they were inspired by image convolutions, but the connection is loose, so don't read too much into it.

The idea is that we start with some node embeddings,

In order to make the mixing trainable, we add a a multiplication by a weight matrix. This matrix applies a linear projection to the mixed embeddings.

The sigmoid activation can also be ReLU or linear. What works best depends on the data.



multiple "layers"

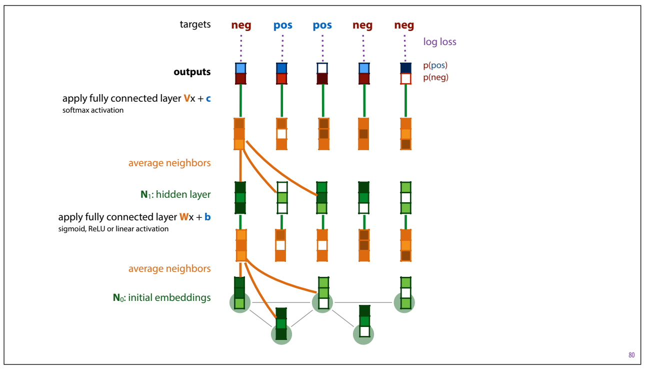
$$\mathbf{N}_1 = \sigma(\mathbf{A}\mathbf{N}_0\mathbf{W})$$

$$\mathbf{O} = \sigma\left(\mathbf{A}\sigma(\mathbf{A}\mathbf{N}_0\mathbf{W})\mathbf{V}\right)$$

after k layers, the embedding mixes in information from k hops away

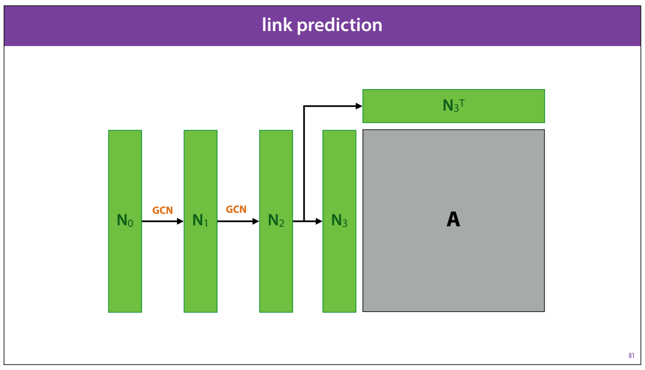
Applying this principle multiple times leads to a multi-layered structure, where we both mix and transform the initial embeddings.

The output **O** is a matrix in which each column represents one of our nodes based both on the initial embedding and the local network structure. We can then use the representations i **O** to perform our classification.



Here is how we do node classification with graph convolutions. We ensure that the embedding size of the last layer is equal to the number of classes (2 in this case). We then apply a softmax activation to these embeddings and interpret them as probabilities over the classes.

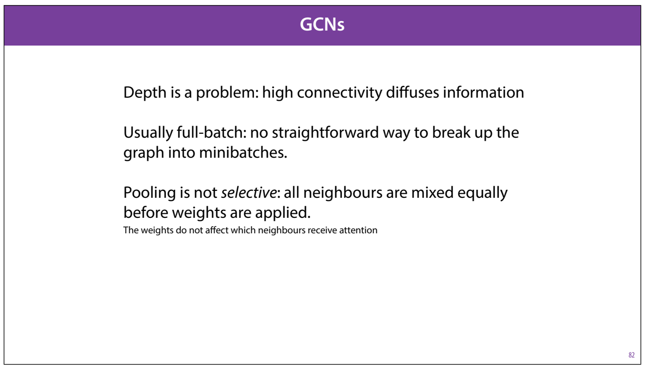
This gives us a full batch of predictions for the whole data, for which we can compute the loss, which we then backpropagate.



link prediction

The mixing trick works for link prediction too. We simply apply a few GCN layers to mix up our embeddings, and then use them to predict our

When we do link prediction, we can perform some graph convolutions on our embeddings and then multiply them out to generate our predictions. We compare these to our training data, and backpropagate the loss.



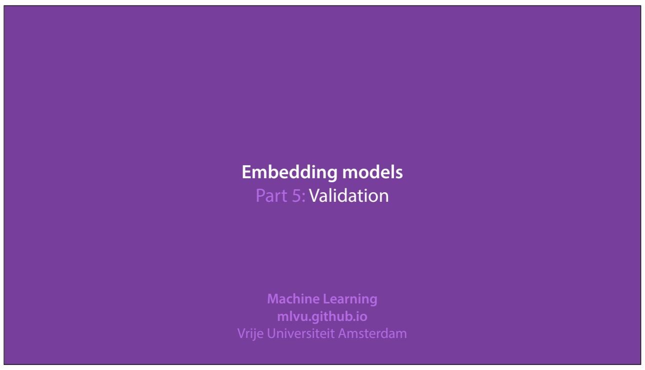
GCNs

Depth is a problem: high connectivity diffuses information

Usually full-batch: no straightforward way to break up the graph into minibatches.

Pooling is not *selective*: all neighbours are mixed equally before weights are applied.
The weights do not affect which neighbours receive attention

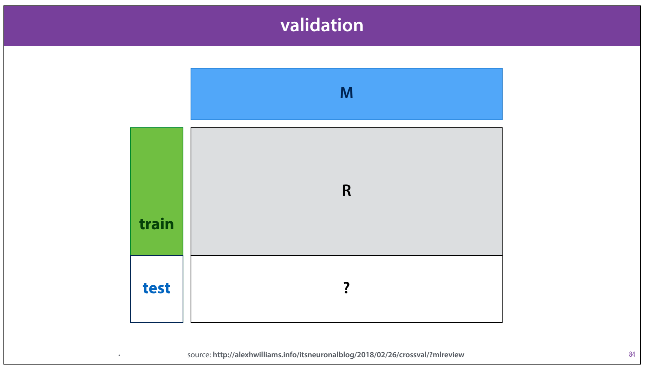
82



Embedding models
Part 5: Validation

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this video, we’ll look at some of the peculiarities of testing a trained embedding model

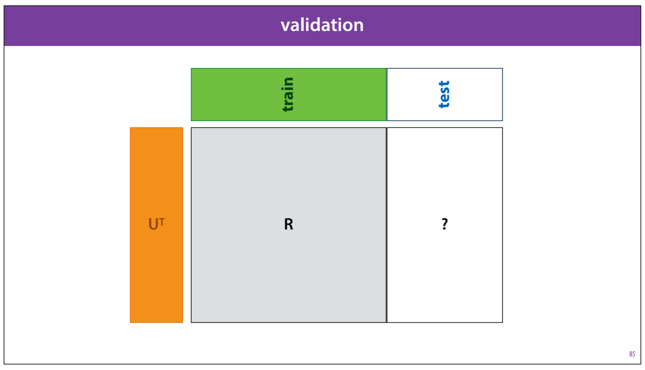


source: http://alexhwilliams.info/itsneuronalblog/2018/02/26/crossval/?mlreview

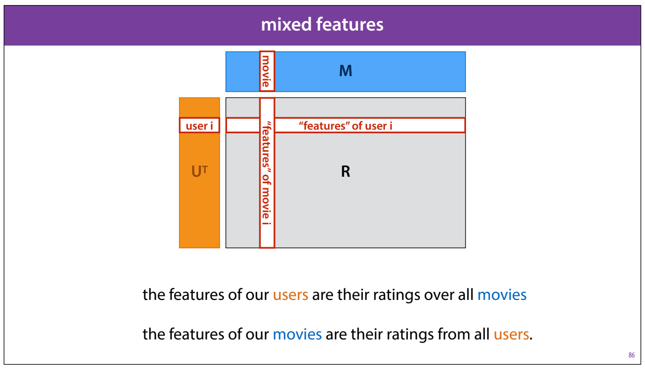
Ot is important to carefully consider our validation protocol. In other words: how do we withhold test data to train on.

Let's start with recommender systems. At first, you might think that it's a good idea to just withhold some users.

However, this doesn't work: if we don't see the users during training, we won't learn embeddings for them, which means we can't generate predictions.

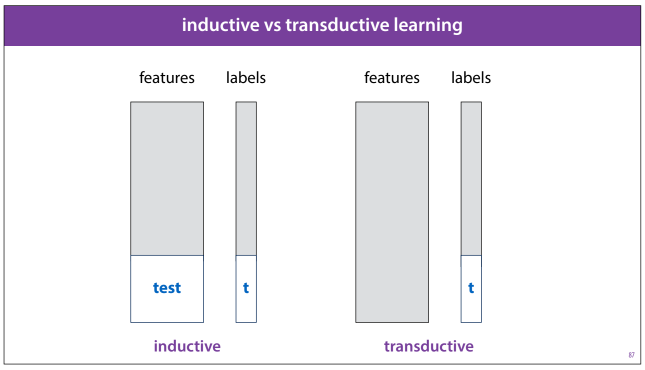


How about if we withhold some movies? The same thing happens.



the features of our users are their ratings over all movies

the features of our movies are their ratings from all users.



This is related to the difference between **inductive** and **transductive** learning. I the transduction setting, the learning is allowed to see the features of all data, but the labels of only the training data.

inductive vs transductive learning

Embedding models only support transductive learning.
If we don't know the objects until after training, we won't have embedding vectors.

Word2Vec: we need to know the *whole* vocabulary at training time.

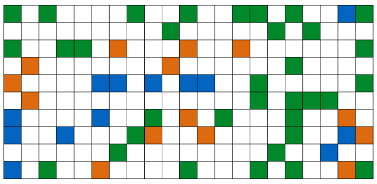
Recommendation: we need to know *all* users and movies.

Graph models: we need to know the *whole* graph.

88

---

recommendation: train, validation, test

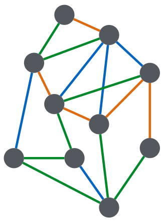


89

To evaluate our matrix factorization, we give the training algorithm all users, and all movies, **but withhold some of the ratings**.

---

graphs: link prediction

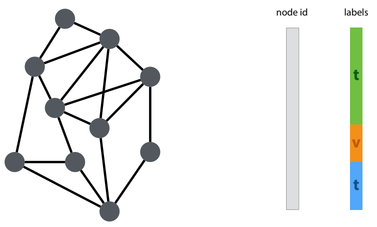


90

The same goes for the links.

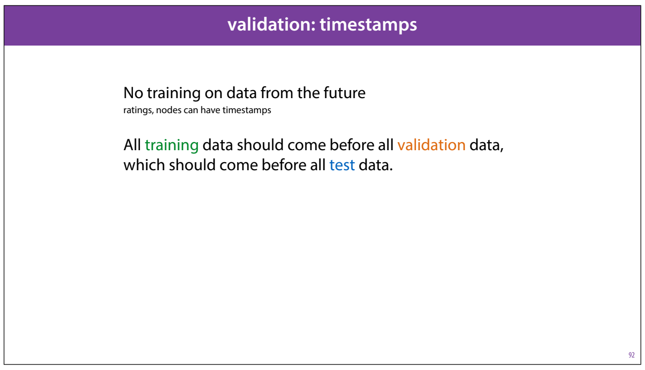
---

graphs: node classification



91

In the case of node classification, we provide the algorithm with the whole graph, and a table linking the node ids to the labels. In this table, we withhold some of the labels.



validation: timestamps

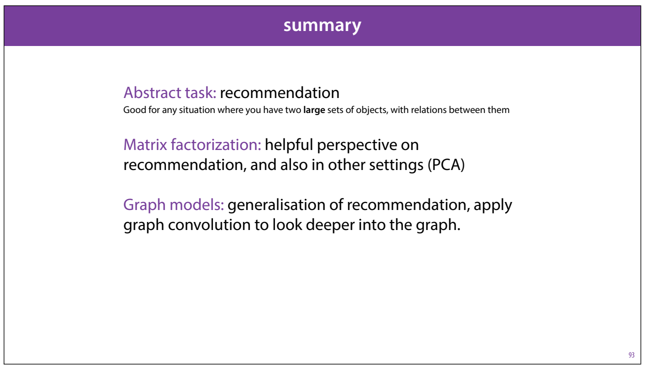
No training on data from the future

ratings, nodes can have timestamps

All training data should come before all validation data, which should come before all test data.

92

If our data has timestamps, we should follow the advice from last lecture as well.



summary

Abstract task: recommendation

Good for any situation where you have two **large** sets of objects, with relations between them

Matrix factorization: helpful perspective on recommendation, and also in other settings (PCA)

Graph models: generalisation of recommendation, apply graph convolution to look deeper into the graph.

93

mlcourse@peterbloem.nl